Contents

Special Files and Protocols (**ADMP**)

intro

introduction to special files and protocols

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip_str.h>
#include <netinet/strioc.h>
```

Description

This section describes various special files and protocols that refer to specific System V STREAMS TCP/IP networking protocol drivers. Features common to a set of protocols are documented as a protocol family.

Protocol Family Entries

A protocol family provides basic services to the protocol implementation to allow it to function within a specific network environment. These services may include packet fragmentation and reassembly, routing, addressing, and basic transport. A protocol family may support multiple methods of addressing, though the current protocol implementations do not. A protocol family is normally comprised of a number of protocols, one per *socket*(2) type. It is not required that a protocol family support all socket types. A protocol family may contain multiple protocols supporting the same socket abstraction.

A protocol supports one of the socket abstractions detailed in **socket**(2). A specific protocol may be accessed by creating a socket of the appropriate type and protocol family, by requesting the protocol explicitly when creating a socket, by executing the appropriate TLI primitives, or by opening the associated STREAMS device.

Protocol Entries

The system currently supports the DARPA Internet protocols. Raw socket interfaces are provided to the IP protocol layer of the DARPA Internet and to the ICMP protocol. Consult the appropriate manual pages in this section for more information.

Routing Ioctls

The network facilities provided limited packet routing. A simple set of data structures comprise a "routing table" used in selecting the appropriate network interface when transmitting packets. This table contains a single entry for each route to a specific network or host. A

user process, the routing daemon, maintains this data base with the aid of two socket-specific *ioctl*(2) commands, SIOCADDRT and SIOCDELRT. The commands allow the addition and deletion of a single routing table entry, respectively. Routing table manipulations may only be carried out by super-user.

A routing table entry has the following form, as defined in *<net/route.h>*:

```
struct rtentry {
        u_long  rt_hash;
        struct  sockaddr rt_dst;
        struct  sockaddr rt_gateway;
        short   rt_flags;
        short   rt_refcnt;
        u_long  rt_use;
        struct  ifnet *rt_ifp;
};
```

with *rt_flags* defined as follows:

```
#define RTF_UP          0x1     /* route usable */
#define RTF_GATEWAY     0x2     /* destination is a gateway */
#define RTF_HOST        0x4     /* host entry (net otherwise) */
#define RTF_DYNAMIC     0x10    /* created dynamically
                                   (by redirect) */
```

Routing table entries are of three general types: those for a specific host, those for all hosts on a specific network, and those for any destination not matched by entries of the first two types (a wildcard route). When the system is booted and addresses are assigned to the network interfaces, each protocol family installs a routing table entry for each interface when it is ready for traffic. Normally the protocol specifies the route through each interface as a "direct" connection to the destination host or network. If the route is direct, the transport layer of a protocol family usually requests the packet be sent to the same host specified in the packet. Otherwise, the interface is requested to address the packet to the gateway listed in the routing entry (that is, the packet is forwarded).

Routing table entries installed by a user process may not specify the hash, reference count, use, or interface fields; these are filled in by the routing routines. If a route is in use when it is deleted (*rt_refcnt* is non-zero), the routing entry will be marked down and removed from the routing table, but the resources associated with it will not be reclaimed until all references to it are released. The routing code returns EEXIST if requested to duplicate an existing entry, ESRCH if requested to delete a non-existent entry, or ENOSR if insufficient resources were available to install a new route. User processes read the routing tables through the */dev/kmem* device. The *rt_use* field contains the number of packets sent along the route.

When routing a packet, the kernel will first attempt to find a route to the destination host. Failing that, a search is made for a route to the network of the destination. Finally, any route to a default ("wildcard") gateway is chosen. If multiple routes are present in the table, the first route found will be used. If no entry is found, the destination is declared to be unreachable.

A wildcard routing entry is specified with a zero destination address value. Wildcard routes are used only when the system fails to find a route to the destination host and network. The combination of wildcard routes and routing redirects can provide an economical mechanism for routing traffic.

Socket Ioctls

There are a few *ioctls* which have significance for the socket layer only. The *ioctl* call has the general form:

```
ioctl(so, code, arg)
```

SIOCPROTO
Enter a socket type into the kernel protocol switch table. The arguments used to create the socket used by this *ioctl* may be zero. The new socket type is downloaded by setting *arg* to a pointer to a specification block with the following structure:

```
struct socknewproto {
        int         family; /* address family (AF_INET, etc.) */
        int         type;   /* protocol type
                               (SOCK_STREAM, etc.) */
        int         proto;  /* per family proto number */
        dev_t       dev;    /* major/minor to use
                               (must be a clone) */
        int         flags;  /* protosw flags */
};
```

The flags currently supported are specified in the *<net/protosw.h>* header file as:

```
        /* exchange atomic messages only */
#define PR_ATOMIC       0x01
        /* addresses given with messages */
#define PR_ADDR         0x02
        /* connection required by protocol */
#define PR_CONNREQUIRED 0x04
#define PR_RIGHTS       0x10    /* passes capabilities */
#define PR_BINDPROTO    0x20    /* pass protocol */
```

SIOCXPROTO
Purge the protocol switch table. The arguments used to create the socket used by this *ioctl* may be zero.

SIOCSPGRP
Set the process group for a socket to enable signaling (SIGUSR1) of that process group when out-of-band data arrives. The argument, *arg*, is a pointer to an **int** and, if positive, is treated as a process ID; otherwise, (if negative) is treated as a process group ID.

SIOCGPGRP
Get the process group ID associated with a particular socket. If the value returned to the **int** location pointed to by *arg* is negative, it should be interpreted as a process group ID; otherwise, it should be interpreted as a process ID.

SIOCCATMARK
Used to ascertain whether or not the socket read pointer is currently at the point (mark) in the data stream where out-of-band data was sent. If a 1 is returned to the *int* location pointed to by *arg*, the next read will return data after the mark. Otherwise (assuming out-of-band data has arrived), the next read will provide data sent by the client prior to transmission of the out-of-band signal.

FIONREAD
Returns (to the **int** location pointed to by *arg*) the number of bytes currently waiting to be read on the socket.

FIONBIO
Toggles the socket into blocking/non-blocking mode. If the **int** location pointed to by *arg* contains a non-zero value, subsequent socket operations that would cause the process to block waiting on a specific event will return abnormally with *errno* set to EWOULDBLOCK; otherwise, the process will block.

Queue Ioctls

Each STREAMS device has default queue high and low water marks, that can be changed by the super-user with the INITQPARMS specification in an **ioctl**(2). The ioctl is done on a driver or module, with the argument being an array of structures of type:

```
struct iocqp {
        ushort iqp_type;
        ushort iqp_value;
}
```

iqp_value specifies the value for the queue parameter according to **iqp_type,** which may be one of: **IQP_RQ**(read queue), **IQP_WQ**(write queue), **IQP_MUXRQ**(mux read queue), **IQP_MUXWQ**(mux write queue), or **IQP_HDRQ**(stream head queue), each OR'ed with either **IQP_LOWAT**(value is for low water mark of queue), or **IQP_HIWAT**(value is for high water mark of queue).

Interface Ioctls

Each network interface in a system corresponds to a path through which messages may be sent and received. A network interface usually has a hardware device associated with it, although certain interfaces such as the loopback interface, *lo*(7), do not.

The following *ioctl* calls may be used to manipulate network interfaces. The *ioctl* is made on a socket (typically of type SOCK_DGRAM) in the desired "communications domain" [see *protocols*(4)]. Unless specified otherwise, the request takes an *ifrequest* structure as its parameter. This structure has the form

```
struct  ifreq {
        char    ifr_name[16]; /* name of interface (e.g. ec0) */
        union {
                struct  sockaddr ifru_addr;
                struct  sockaddr ifru_dstaddr;
                struct  sockaddr ifru_broadaddr;
                short   ifru_flags;
                int     ifru_metric;
                struct onepacket ifru_onepacket;
        } ifr_ifru;
#define ifr_addr        ifr_ifru.ifru_addr        /* address */
                /* other end of p-to-p link */
#define ifr_dstaddr     ifr_ifru.ifru_dstaddr
                /* broadcast address */
#define ifr_broadaddr   ifr_ifru.ifru_broadaddr
#define ifr_flags       ifr_ifru.ifru_flags       /* flags */
                /* routing metric */
#define ifr_metric      ifr_ifru.ifru_metric
                /* one-packet mode params */
#define ifr_onepacket   ifr_ifru.ifru_onepacket
};
```

SIOCSIFADDR
Set interface address for protocol family. Following the address assignment, the "initialization" routine for the interface is called.

SIOCGIFADDR
Get interface address for protocol family.

SIOCSIFDSTADDR
Set point to point address for protocol family and interface.

SIOCGIFDSTADDR
Get point to point address for protocol family and interface.

SIOCSIFBRDADDR
Set broadcast address for protocol family and interface.

SIOCGIFBRDADDR
Get broadcast address for protocol family and interface.

SIOCSIFFLAGS
Set interface flags field. If the interface is marked down, any processes currently routing packets through the interface are notified; some interfaces may be reset so that incoming packets are no longer received. When marked up again, the interface is reinitialized.

SIOCGIFFLAGS
Get interface flags.

SIOCSIFMETRIC
Set interface routing metric. The metric is used only by user-level routers.

SIOCGIFMETRIC
Get interface metric.

SIOCSIFONEP
Set one-packet mode parameters. The *ifr_onepacket* field of the *ifreq* structure is used for this request. This structure is defined as follows:

```
struct onepacket {
        int     spsize;     /* small packet size */
        int     spthresh;   /* small packet threshold */
};
```

One-packet mode is enabled by setting the IFF_ONEPACKET flag (see SIOCSIFFLAGS above). See *tcp*(7) for an explanation of one-packet mode.

SIOCGIFONEP
Get one-packet mode parameters.

SIOCGIFCONF
Get interface configuration list. This request takes an *ifconf* structure (see below) as a value-result parameter. The *ifc_len* field should be initially set to the size of the buffer pointed to by *ifc_buf*. On return it will contain the length, in bytes, of the configuration list.

```
    /* Structure used in SIOCGIFCONF request.
     * Used to retrieve interface configuration
     * for machine (useful for programs which
     * must know all networks accessible).
     */
struct  ifconf {
        /* size of associated buffer */
        int     ifc_len;
```

```
        union {
                caddr_t ifcu_buf;
                struct  ifreq *ifcu_req;
        } ifc_ifcu;
/* buffer address */
#define ifc_buf ifc_ifcu.ifcu_buf
/* array of structures returned */
#define ifc_req ifc_ifcu.ifcu_req
};
```

Streams Ioctl Interface

Socket *ioctl* calls can also be issued using STREAMS file descriptors. The standard *strioctl* structure is used, with the *ic_cmd* field containing the socket *ioctl* code (from **<sys/socket.h>**) and the *ic_db* field pointing to the data structure appropriate for that *ioctl*, for all socket *ioctl*s except SIOCGIFCONF. For the SIOCGIFCONF *ioctl*, an *ifconf* structure is not used. Rather, the *ic_db* field points to the buffer to receive the *ifreq* structures.

TLI Options Management

Options may be set and retrieved in a manner similar to *getsockopt* (2) and *setsockopt* (2) using *t_optmgmt* (3N). Options are communicated using an options buffer, which contains a list of options. Each option consists of an option header and an option value. The *opthdr* structure gives the format of the option header:

```
struct opthdr {
        long level;     /* protocol level affected */
        long name;      /* option to modify */
        long len;       /* length of option value (in bytes) */
};
```

The option value must be a multiple of sizeof(long) bytes in length, and must immediately follow the option header. Following the option value is the header of the next option, if present.

To get the values of options, set the *flags* field of the *t_optmgmt* structure to T_CHECK. It is not necessary to set the *len* fields in the option headers to the expected lengths of the option values, nor is it necessary to provide space between option headers for the option values to be stored (the *len* fields should be set to zero and the option headers should be adjacent). A new options buffer will be formatted and returned to the user. Note that T_CHECK may have failed even if *t_optmgmt* returns zero. The user must check the *flags* field of the returned *t_optmgmt* structure. If this field contains T_FAILURE, one or more of the options were invalid.

To set options, set the *flags* field of the *t_optmgmt* structure to T_NEGOTIATE.

To retrieve the default values of all options, set the *flags* field of the *t_optmgmt* structure to T_DEFAULT. For this operation, no input buffer should be specified.

Note

System V STREAMS TCP/IP man pages frequently cite appropriate RFCs (Requests for Comments). RFCs can be obtained from the DDN Network Information Center, SRI International, Menlo Park, CA 94025.

See Also

ioctl(SSC), socket(SSC), t_optmgmt(NSL), tcp(ADMP).

arp

Address Resolution Protocol

Description

ARP is a protocol used to map dynamically between DARPA Internet and 10Mb/s Ethernet addresses. It is used by all the 10Mb/s Ethernet interface drivers running the Internet protocols.

ARP caches Internet-Ethernet address mappings. When an interface requests a mapping for an address not in the cache, ARP queues the message which requires the mapping and broadcasts a message on the associated network requesting the address mapping. If a response is provided, the new mapping is cached and any pending message is transmitted. ARP will queue at most one packet while waiting for a mapping request to be answered; only the most recently "transmitted" packet is kept. The ARP protocol is implemented by a STREAMS driver to do the protocol negotiation, and by a separate STREAMS module to do the address translation.

To facilitate communications with systems that do not use ARP, *ioctl* s are provided to enter and delete entries in the Internet-to-Ethernet tables. Usage:

```
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <net/if.h>
struct arpreq arpreq;

ioctl(s, SIOCSARP, (caddr_t)&arpreq);
ioctl(s, SIOCGARP, (caddr_t)&arpreq);
ioctl(s, SIOCDARP, (caddr_t)&arpreq);
```

Each *ioctl* takes the same structure as an argument. SIOCSARP sets an ARP entry, SIOCGARP gets an ARP entry, and SIOCDARP deletes an ARP entry. These *ioctls* may be applied to any socket descriptor *s*, but only by the superuser. The *arpreq* structure is as follows:

```
   /* ARP ioctl request */
struct arpreq {
   struct sockaddr      arp_pa;/* protocol address */
   struct sockaddr      arp_ha;/* hardware address */
   int                  arp_flags;/* flags */
};
   /*  arp_flags field values */
#defineATF_COM       0x02/* completed entry
                            (arp_ha valid) */
#defineATF_PERM      0x04/* permanent entry */
```

```
#defineATF_PUBL       0x08/* publish
                              (respond for other host) */
#defineATF_USETRAILERS0x10/* send trailer packets
                              to host */
```

The address family for the *arp_pa sockaddr* , must be AF_INET; for the *arp_ha sockaddr* it must be AF_UNSPEC. The only flag bits which may be written are ATF_PERM, ATF_PUBL and ATF_USETRAILERS. ATF_PERM causes the entry to be permanent if the *ioctl* call succeeds. The peculiar nature of the ARP tables may cause the *ioctl* to fail if more than 8 (permanent) Internet host addresses hash to the same slot. ATF_PUBL specifies that the ARP code should respond to ARP requests for the indicated host coming from other machines. This allows a host to act as an "ARP server," which may be useful in convincing an ARP-only machine to talk to a non-ARP machine.

ARP can also negotiate the use of trailer IP encapsulations; trailers are an alternate encapsulation used to allow efficient packet alignment for large packets despite variable-sized headers. Hosts that wish to receive trailer encapsulations indicate so by sending gratuitous ARP translation replies along with replies to IP requests; they are also sent in reply to IP translation replies. The negotiation is thus fully symmetrical, in that either or both hosts may request trailers. The ATF_USETRAILERS flag is used to record the receipt of such a reply, and enables the transmission of trailer packets to that host.

ARP watches passively for hosts impersonating the local host (that is, a host that responds to an ARP mapping request for the local host's address).

Diagnostics

duplicate IP address!! sent from ethernet address: %x:%x:%x:%x:%x:%x. ARP has discovered another host on the local network that responds to mapping requests for its own Internet address.

Files

/dev/inet/arp

See Also

arp(ADMP), ifconfig(ADMN), inet(ADMP).

e3A

3C501 Ethernet Driver

Description

The e3A driver provides an LLI interface to a 3Com 3C501 ethernet card. As with other network interfaces, e3A interface must have network addresses assigned for each address family with which it is to be used. (Currently, only the Internet address family is supported.) These addresses may be set or changed with the SIOCSIFADDR ioctl.

Files

/dev/e3A[0-3]

See Also

intro(ADMP), inet(ADMP).

e3B

3C503 Ethernet Driver

Description

The e3B driver provides an LLI interface to a 3Com 3C503 ethernet card. As with other network interfaces, e3B interface must have network addresses assigned for each address family with which it is to be used. (Currently, only the Internet address family is supported.) These addresses may be set or changed with the SIOCSIFADDR ioctl.

Files

/dev/e3B[0-3]

See Also

intro(ADMP), inet(ADMP).

eli

EMD convergence module

Description

Eli acts as a convergence module between the EMD Ethernet Driver, and another STREAMS driver or module. *Eli* provides an LLI compatible interface, which is expected by *ip*(ADMP). *Eli* must be pushed on the STREAM between *ip* and *emd*.

It is expected that since the 10base5 driver is now available as a product, EMD will no longer be used, and *eli* will become obsolete.

See Also

strcf(SFF), ip(ADMP).

icmp

Internet Control Message Protocol

Syntax

#include <sys/socket.h>
#include <netinet/in.h>

s = socket(AF_INET, SOCK_RAW, proto);

Description

ICMP is the error and control message (or device) protocol used by IP and the Internet protocol family. It may be accessed through a "raw socket" for network monitoring and diagnostic functions. The *proto* parameter to the socket call to create an ICMP socket is obtained from *getprotobyname* . [See *getprotoent* (SLIB).] ICMP sockets are connectionless, and are normally used with the *sendto* and *recvfrom* calls; the *connect* (SSC) call may also be used to fix the destination for future packets (in which case the *read* (S) or *recv* (SSC) and *write* (S) or *send* (SSC) system calls may be used).

Outgoing packets automatically have an IP header prepended to them (based on the destination address). Incoming packets are received with the IP header and options intact.

Diagnostics

A socket operation may fail with one of the following errors returned:

[EISCONN] when trying to establish a connection on a socket that already has one, or when trying to send a datagram with the destination address specified and the socket already connected;

[ENOTCONN] when trying to send a datagram, but no destination address is specified, and the socket has not been connected;

[ENOSR] when the system runs out of memory for an internal data structure;

[EADDRNOTAVAIL] when an attempt is made to create a socket with a network address for which no network interface exists.

Files

/dev/inet/icmp

See Also

send(SSC), recv(SSC), intro(ADMP), inet(ADMP), ip(ADMP).

inet

Internet protocol family

Syntax

#include <sys/types.h>
#include <netinet/in.h>

Description

The Internet protocol family is a set of protocols using the *Internet Protocol* (IP) network layer and the Internet address format. The Internet family provides protocol support for the SOCK_STREAM, SOCK_DGRAM, and SOCK_RAW socket types; the SOCK_RAW interface provides access to the IP protocol.

Addressing

Internet addresses are four-byte quantities, stored in network standard format. The include file < **sys/in.h** > defines this address as a discriminated union.

Sockets bound to the Internet protocol family use the following addressing structure:

```
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

When using sockets, the *sin_family* is specified in host order, and the sin_port and *sin_addr* fields are specified in network order.

Sockets may be created with the local address INADDR_ANY to affect wildcard matching on incoming messages. The address in a *connect*(SSC) or *sendto* [see *send*(SSC)] call may be given as INADDR_ANY to mean "this host." The distinguished address INADDR_BROADCAST is allowed as a shorthand for the broadcast address on the primary network if the first network configured supports broadcast.

When using the Transport Layer Interface (TLI), transport providers such as *tcp*(ADMP) support addresses whose lengths vary from eight to sixteen bytes. The eight byte form is the same as a *sockaddr_in* without the *sin_zero* field. The sixteen byte form is identical to

sockaddr_in. Additionally, when using TLI, the *sin_family* field is accepted in either host or network order.

Protocols

The Internet protocol family is comprised of the IP transport protocol, Internet Control Message Protocol (ICMP), Transmission Control Protocol (TCP), and User Datagram Protocol (UDP). TCP is used to support the SOCK_STREAM abstraction; UDP is used to support the SOCK_DGRAM abstraction. A raw interface to IP is available by creating an Internet socket of type SOCK_RAW. The ICMP message protocol is accessible from a raw socket.

The 32-bit Internet address contains both network and host parts. It is frequency-encoded; the most significant bit is clear in Class A addresses, in which the high-order 8 bits are the network number. Class B addresses use the high-order 16 bits as the network field, and Class C addresses have a 24-bit network part. Sites with a cluster of local networks and a connection to the DARPA Internet may choose to use a single network number for the cluster; this is done by using subnet addressing. The local (host) portion of the address is further subdivided into subnet and host parts. Within a subnet, each subnet appears to be an individual network; externally, the entire cluster appears to be a single, uniform network requiring only a single routing entry. Subnet addressing is enabled and examined by the following *ioctl*(S) commands on a datagram socket in the Internet "communications domain"; they have the same form as the SIOCIFADDR command. [See *intro*(ADMP).]

SIOCSIFNETMASK
Set interface network mask. The network mask defines the network part of the address; if it contains more of the address than the address type would indicate, then subnets are in use.

SIOCGIFNETMASK
Get interface network mask.

See Also

ioctl(S), socket(SSC), intro(ADMP), intro(SFF), icmp(ADMP), ip(ADMP), tcp(ADMP), udp(ADMP).

Note

The Internet protocol support is subject to change as the Internet protocols develop. Users should not depend on details of the current implementation, but rather the services exported.

ip

Internet Protocol

Syntax

```
#include <sys/socket.h>
#include <netinet/in.h>

s = socket(AF_INET, SOCK_RAW, proto);
```

Description

IP is the network layer protocol used by the Internet protocol family. Options may be set at the IP level when using higher-level protocols that are based on IP (such as TCP and UDP). It may also be accessed through a "raw socket" or device when developing new protocols or special purpose applications.

A single generic option IP_OPTIONS, is supported at the IP level, and may be used to provide IP options to be transmitted in the IP header of each outgoing packet. Options are set with *setsockopt* and examined with *getsockopt* . [See *getsockopt*(SSC).] The format of IP options to be sent is that specified by the IP protocol specification, with one exception: the list of addresses for Source Route options must include the first-hop gateway at the beginning of the list of gateways. The first-hop gateway address will be extracted from the option list and the size adjusted accordingly before use. IP options may be used with any socket type in the Internet family.

Raw IP sockets are connectionless, and are normally used with the *sendto* and *recvfrom* calls; the *connect*(SSC) call may also be used to fix the destination for future packets (in which case, the *read*(S) or *recv*(SSC), and *write*(S) or *send*(SSC) system calls may be used).

If *proto* is 0, the default protocol IPPROTO_RAW is used for outgoing packets, and only incoming packets destined for that protocol are received. If *proto* is non-zero, that protocol number will be used on outgoing packets and to filter incoming packets. *Proto* must be specified in *sockcf(SFF)*.

Outgoing packets automatically have an IP header prepended to them (based on the destination address given and the protocol number the socket is created with). Incoming packets are received with IP header and options intact.

Diagnostics

A socket operation may fail with one of the following errors returned:

[EISCONN] when trying to establish a connection on a socket which already has one, or when trying to send a datagram with the destination address specified and the socket already connected;

[ENOTCONN] when trying to send a datagram, but no destination address is specified, and the socket has not been connected;

[ENOSR] when the system runs out of memory for an internal data structure;

[EADDRNOTAVAIL] when an attempt is made to create a socket with a network address for which no network interface exists.

The following errors specific to IP may occur when setting or getting IP options:

[EINVAL] An unknown socket option name was given.

[EINVAL] The IP option field was improperly formed; an option field was shorter than the minimum value or longer than the option buffer provided.

Files

/dev/inet/ip
/dev/inet/rip

See Also

getsockopt(SSC), send(SSC), recv(SSC), sockcf(SFF), intro(ADMP), icmp(ADMP), inet(ADMP).

llcloop

software loopback network interface

Syntax

```
#include <sys/socket.h>
#include <netinet/in.h>
struct sockaddr_in sin;

s = socket(AF_INET, SOCK_XXX, 0);
.
.
.
sin.sin_addr.s_addr = hton1 (INADDR_ANY);
bind(s, (char *)&sin, sizeof(sin));
```

Description

The *llcloop* interface is a software loopback mechanism which may be used for performance analysis, software testing, and/or local communication. As with other network interfaces, the loopback interface must have network addresses assigned for each address family with which it is to be used. (Currently, only the Internet address family is supported.) These addresses may be set or changed with the SIOCSIFADDR ioctl. The loopback interface should be the first one configured, otherwise nameserver lookups for hostnames of other interfaces may fail.

Files

/dev/llcloop

See Also

intro(ADMP), inet(ADMP).

slip

serial line IP network interface

Description

The slip interface is a driver that allows IP datagrams to be sent over normal serial lines. This is useful for connecting machines that do not have Ethernet hardware. As with other network interfaces, the slip interface must have network addresses assigned for each address family with which it is to be used. (Currently, only the Internet address family is supported.) These addresses may be set or changed with the SIOCSIFADDR ioctl.

See Also

ifconfig(ADMN), slattach(ADMN), sldetach(ADMN), intro(ADMP), inet(ADMP).

sock

Socket Interface Driver

Description

The socket driver is used to provide socket emulation to applications. Sockets are an alternate entry point into transport providers, such as *tcp*(ADMP). The socket driver is a character device that acts as an alternate stream head, augmenting the functions of the standard stream head. It also provides support for miscelleanous functions such as *select*(SSC).

FILES

/dev/socksys

SEE ALSO

ifconfig(ADMN), intro(SSC), slattach(ADMN), sldetach(ADMN), intro(ADMP), inet(ADMP)

tcp

Internet Transmission Control Protocol

Syntax

```
#include <sys/socket.h>
#include <netinet/in.h>

s = socket(AF_INET, SOCK_STREAM, 0);
```

Description

The TCP protocol provides reliable, flow-controlled, two-way transmission of data. It is a byte-stream protocol used to support the SOCK_STREAM abstraction. TCP uses the standard Internet address format and, in addition, provides a per-host collection of "port addresses." Thus, each address is composed of an Internet address specifying the host and network, with a specific TCP port on the host identifying the peer entity.

Sockets using the *tcp* protocol are either "active" or "passive." Active sockets initiate connections to passive sockets. By default, TCP sockets are created active; to create a passive socket, the *listen*(SSC) system call must be used after binding the socket with the *bind*(SSC) system call. Only passive sockets may use the *accept*(SSC) call to accept incoming connections. Only active sockets may use the *connect*(SSC) call to initiate connections.

Passive sockets may "underspecify" their location to match incoming connection requests from multiple networks. This technique, called "wildcard addressing," allows a single server to provide service to clients on multiple networks. To create a socket that listens on all networks, the Internet address INADDR_ANY must be bound. The TCP port may still be specified at this time; if the port is not specified, the system will assign one. Once a connection has been established, the socket's address is fixed by the peer entity's location. The address assigned the socket is the address associated with the network interface through which packets are being transmitted and received. Normally, this address corresponds to the peer entity's network.

TCP supports one socket option that is set with *setsockopt* and tested with *getsockopt* . [See *getsockopt*(SSC).] Under most circumstances, TCP sends data when it is presented; when outstanding data has not yet been acknowledged, it gathers small amounts of output to be sent in a single packet once an acknowledgment is received. For a small number of clients, such as window systems that send a stream of mouse events that receive no replies, this packetization may cause significant delays. Therefore, TCP provides a boolean option,

TCP_NODELAY (from *<netinet/tcp.h>*, to defeat this algorithm. The option level for the *setsockopt* call is the protocol number for TCP, available from *getprotobyname* . [See *getprotoent* (SLIB).]

Options at the IP transport level may be used with TCP; see *ip* (ADMP). Incoming connection requests that are source-routed are noted, and the reverse source route is used in responding.

TCP is also available as a TLI connection-oriented protocol via the special file **/dev/inet/tcp**. TCP options are supported via the TLI options mechanism.

TCP provides a facility, *one-packet mode*, that attempts to improve performance over Ethernet interfaces that cannot handle back-to-back packets. One-packet mode may be set by *ifconfig* (1M) for such an interface. On a connection that uses an interface for which one-packet mode has been set, TCP attempts to prevent the remote machine from sending back-to-back packets by setting the window size for the connection to the maximum segment size for the interface.

Certain TCP implementations have an internal limit on packet size that is less than or equal to half the advertised maximum segment size. When connected to such a machine, setting the window size to the maximum segment size would still allow the sender to send two packets at a time. To prevent this, a "small packet size" and a "small packet threshold" may be specified when setting one-packet mode. If, on a connection over an interface with one-packet mode enabled, TCP receives a number of consecutive packets of the small packet size equal to the small packet threshold, the window size is set to the small packet size.

Diagnostics

A socket operation may fail with one of the following errors returned:

[EISCONN] when trying to establish a connection on a socket which already has one;

[ENOSR] when the system runs out of memory for an internal data structure;

[ETIMEDOUT] when a connection was dropped due to excessive retransmissions

[ECONNRESET] when the remote peer forces the connection to be closed;

[ECONNREFUSED] when the remote peer actively refuses connection establishment (usually because no process is listening to the port);

TCP (ADMP)

[EADDRINUSE] when an attempt is made to create a socket with a port which has already been allocated;

[EADDRNOTAVAIL] when an attempt is made to create a socket with a network address for which no network interface exists.

Files

/dev/inet/tcp

See Also

ifconfig(ADMN), getsockopt(SSC), socket(SSC), intro(ADMP), inet(ADMP), ip(ADMP).

UDP (ADMP)

udp

Internet User Datagram Protocol

Syntax

```
#include <sys/socket.h>
#include <netinet/in.h>

s = socket(AF_INET, SOCK_DGRAM, 0);
```

Description

UDP is a simple, unreliable datagram protocol that is used to support the SOCK_DGRAM abstraction for the Internet protocol family. UDP sockets are connectionless, and are normally used with the *sendto* and *recvfrom* calls; the *connect*(SSC) call may also be used to fix the destination for future packets (in which case, the *recv*(SSC), or *read*(S) and *send*(SSC), or *write*(S) system/library calls may be used). In addition, UDP is available as TLI connectionless transport via the special file **/dev/inet/udp**.

UDP address formats are identical to those used by TCP. In particular, UDP provides a port identifier in addition to the normal Internet address format. Note that the UDP port space is separate from the TCP port space (that is, a UDP port may not be "connected" to a TCP port). In addition, broadcast packets may be sent (assuming the underlying network supports this) by using a reserved broadcast address; this address is network interface-dependent.

Options at the IP transport level may be used with UDP; see *ip*(ADMP).

Diagnostics

A socket operation may fail with one of the following errors returned:

[EISCONN] when trying to establish a connection on a socket which already has one, or when trying to send a datagram with the destination address specified and the socket already connected;

[ENOTCONN] when trying to send a datagram, but no destination address is specified, and the socket has not been connected;

[ENOSR] when the system runs out of memory for an internal data structure;

[EADDRINUSE] when an attempt is made to create a socket with a port that has already been allocated;

[EADDRNOTAVAIL] when an attempt is made to create a socket with a network address for which no network interface exists.

Files

/dev/inet/udp

See Also

getsockopt(SSC), recv(SSC), send(SSC), socket(SSC), intro(ADMP), inet(ADMP), ip(ADMP), RFC768.

vty

pseudo terminal slave driver

ttyp

pseudo terminal master driver

Description

The ttyp and vty drivers together provide support for a device-pair termed a *pseudo terminal*. A pseudo terminal is a pair of character devices, a *master* device and a *slave* device. The slave device provides processes an interface identical to that described in *termio*(ADMP). However, whereas all other devices which provide the interface described in *termio*(ADMP) have a hardware device of some sort behind them, the slave device has, instead, another process manipulating it through the master half of the pseudo terminal. That is, anything written on the master device is given to the slave device as input and anything written on the slave device is presented as input on the master device.

The following *ioctl* call applies only to pseudo terminals:

TIOCPKT
: Enable/disable *packet* mode. Packet mode is enabled by specifying (by reference) a nonzero parameter and disabled by specifying (by reference) a zero parameter. When applied to the master side of a pseudo terminal, each subsequent *read* from the terminal will return data written on the slave part of the pseudo terminal preceded by a zero byte (symbolically defined as TIOCPKT_DATA), or a single byte reflecting control status information. In the latter case, the byte is an inclusive-or of zero or more of the bits:

  TIOCPKT_FLUSHREAD
  : whenever the read queue for the terminal is flushed.

  TIOCPKT_FLUSHWRITE
  : whenever the write queue for the terminal is flushed.

  TIOCPKT_STOP
  : whenever output to the terminal is stopped a la ^S.

  TIOCPKT_START
  : whenever output to the terminal is restarted.

  TIOCPKT_DOSTOP
  : whenever *t_stopc* is ^S and *t_startc* is ^Q.

VTY (ADMP)

TIOCPKT_NOSTOP
whenever the start and stop characters are not ^S/^Q.

While this mode is in use, the presence of control status information to be read from the master side may be detected by a *select* for exceptional conditions.

This mode is used by *rlogin*(TC) and *rlogind*(ADMN) to implement a remote-echoed, locally ^S/^Q flow-controlled remote login with proper back-flushing of output; it can be used by other similar programs.

Files

/dev/ptyp[0-f][0-f]	master pseudo terminals
/dev/ttyp[0-f][0-f]	slave pseudo terminals

See Also

termio(ADMP).